To analyze the order of growth of the running time for the given functions, we need to examine how the running time of each function scales with respect to the length of the input string `s`.

Function `reverse1`

```java
public static String reverse1(String s) {
    int n = s.length();
    String reverse = "";
    for (int i = 0; i < n; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}
```

1. The `for` loop runs `n` times, where `n` is the length of the string `s`.
2. Within each iteration of the loop, a new string is created by concatenating `s.charAt(i)` and `reverse`. Concatenating two strings of length `a` and `b` takes $O(a + b)$ time.
3. Initially, `reverse` is empty, but it grows by one character in each iteration, so the time complexity for the `i`th iteration is $O(i)$.

The total time complexity is the sum of the complexities of all iterations: $O(1) + O(2) + O(3) + \cdots + O(n)$

This sum is an arithmetic series that adds up to: $O(n^2)$

Therefore, the time complexity of `reverse1` is $O(n^2)$.

Function `reverse2`

```java
public static String reverse2(String s) {
    int n = s.length();
    if (n <= 1) return s;
    String left = s.substring(0, n/2);
    String right = s.substring(n/2, n);
    return reverse2(right) + reverse2(left);
}
```

1. The function splits the string into two halves and calls itself recursively on each half.
2. The `substring` operation takes $O(n)$ time, and each call to `reverse2` makes two recursive calls on strings of approximately half the original length.
3. The concatenation operation `reverse2(right) + reverse2(left)` takes $O(n)$ time because the lengths of `reverse2(right)` and `reverse2(left)` add up to `n`.

To express the time complexity, we use the recurrence relation:
$T(n) = 2T(n/2) + O(n)$

Using the Master Theorem for divide-and-conquer recurrences: $T(n) = aT(n/b) + f(n)$ where $a = 2$, $b = 2$, and $f(n) = O(n)$.

The Master Theorem states:

- If $f(n) = O(n^c)$ where $c < \log_b a$, then $T(n) = O(n^{\log_b a})$.
- If $f(n) = O(n^c)$ where $c = \log_b a$, then $T(n) = O(n^c \log n)$.
- If $f(n) = O(n^c)$ where $c > \log_b a$, then $T(n) = O(f(n))$.

In this case: $a = 2, b = 2, \log_b a = \log_2 2 = 1$

Since $f(n) = O(n)$ and $c = 1$, which equals $\log_b a$, the second case of the Master Theorem applies: $T(n) = O(n \log n)$

Therefore, the time complexity of `reverse2` is $O(n \log n)$.

Summary

- `reverse1` has a time complexity of $O(n^2)$.
- `reverse2` has a time complexity of $O(n \log n)$.

This shows that `reverse2` is more efficient than `reverse1` for large input sizes.